Databases II: DB programming

Stored Procedures – Triggers

Tables and user defined types

Dynamic SQL

What do you learn in this chapter ?

- How the relational model has been extended to support advance database applications
- procedural SQL statements
- stored procedures and stored functions
- triggers
- Advanced datatypes
- Dynamic SQL

SQL as complete language (PSM)

Persistent Stored Modules

- Originally SQL was not a complete programming language
  - But SQL could be embedded in a standard programming language
- SQL/PSM, as a complete programming language
  - variables, constants, datatypes
  - operators
  - Controle structurs
    - if, case, while, for, …
  - procedures, functions
  - exception handling
- PSM = stored procedures and stored functions
- Examples
  - SQL Server: Transact SQL
  - Oracle: PL/SQL
  - DB2: SQL PL

procedural SQL extensions are **vendor specific**;

database systems that use these extensions are more **difficult to migrate** to another RDBMS.

Proprietary languages!

ORACLE PL/SQL

```
create procedure youngest (nameofyoungest out varchar2(20))
as
begin
    select name into nameofyoungest from players
    where birth_date  = (select max(birth_date) from players);
end;
```

MICROSOFT TRANSACT-SQL

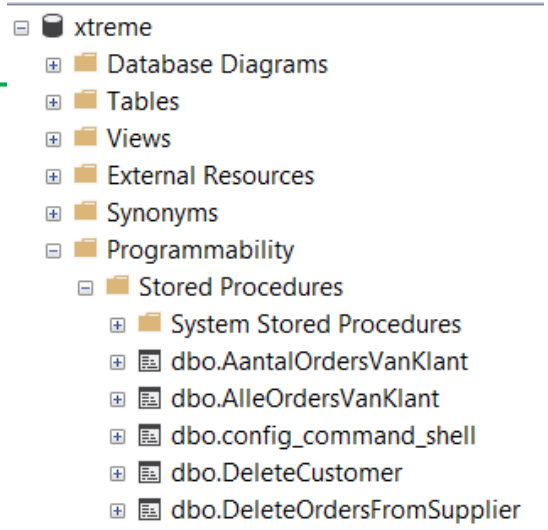
```
create procedure youngest @nameofyoungest varchar(20) out
as
select @ nameofyoungest = name from players
where birth_date = (select max(birth_date) from players)
```

mySQL

```
create procedure youngest (out nameofyoungest varchar(20))
begin
set nameofyoungest =(select name from players
where birth_date = (select max(birth_date) from players)
end
```

Stored Procedures and Stored Functions

Example: delete a customer




```
create procedure DeleteCustomer @custno int = NULL
as

if @custno IS NULL
begin
    print 'Please provide a customerid'
    return
end

if not exists (select NULL from customer where customerid=@custno)
begin
    print 'The customer doesn''t exist.'
    return
end

if exists (select NULL from orders where customerid=@custno)
begin
    print 'The customer already has orders and can''t be deleted.'
    return
end

delete from customer where customerid = @custno
print 'The customer has been succesfully deleted'
```

Stored Procedures

- Call in MS SQL Server:
  - `exec DeleteCustomer 3000`

- As DDL-element (table, index, view, …) stored in the DB

Why using PSM's (SP and UDF)?

PSM vs. 3GL (Java, .NET, C++, Cobol…)

- Older versions of SQL Server (6.5 & 7) and Oracle:
  - query-optimisation and execution plan caching & reuse (see further) is better when using PSM

  →SQL execution through PSM was more performant
- now: +/- same optimisation, no matter how query arrives at database.
- unfortunately performance is still used as a reason to use PSM's.

PSM: advantages

- code modularisation
  - reduce redundant code: put frequently used queries in SP and reuse in 3GL

  →Less maintenance tasks at schema updates.

  →Often used for CRUD-operations
- customisation of "closed" systems like ERP: via stored procedures and triggers you can influence behaviour
- security
  - Exclude direct queries on tables
  - SP's determine what is allowed
  - avoid SQL-injection attacks by using input parameters
- central administration of (parts of) DB code

PSM-disadvantages

- Reduced scalability: business logic and db processing run on same server, can cause bottlenecks.
- Vendor lock-in
  - syntax = non standard: port from ex. MS SQL Server to Oracle very time consuming
  - but portability comes with a price (ex. built-in functions can't be used)
- Two programming languages:
  1. JAVA/.NET/…..
  2. SP / UDF
- Two debug environments
- SP/UDF: limited OO support

Rules of thumb

- Avoid PSM for larger business logic
- Consider using PSM for technical stuff
  - logging/auditing/validation
- Make choice portability vs. vendor lock-in taking into account
  - your business departments
  - corporate IT policies

stored procedure

a stored procedure is a named collection of SQL and control-of-flow commands (program) that is stored as a database object

- what?
  - Analogous to procedures/functions in other languages
  - Can be called from a program, trigger or stored procedure
  - Is saved in the catalogue
  - Accepts input and output parameters
  - Returns status information about the correct or incorrect execution of the stored procedure
  - Contains the tasks to be executed

VARIABLES

Local variables

- A variable name always starts with a @

```
DECLARE @variable_name1 data_type [, @variable_name2
data_type ...]
```

- Assign a value to a variable

```
SET @variable_name = expression
SELECT @variable_name = column_specification
```

- Set and select are equivalent, but set is ANSI standard

```
declare @max decimal(7,2)
set @max = (select max(orderamount) from orders)
select @max = max(orderamount) from orders
print @max
```

- With select you can assign a value to several variables at once:

```
declare @max decimal(7,2),@nrOfOrders int
select @max = max(orderamount), @nrOfOrders = count(*) from orders
print @max
print @nrOfOrders
```

Local variables

```
PRINT string_expression
```

Give message to user

- PRINT: SQL Server Management Studio shows a message in the message tab

- As an alternative you can also use select

```
declare @max decimal(7,2),@nrOfOrders int
select @nrOfOrders = count(*) from orders
select 'The number of orders is: ' + str(@nrOfOrders)
```

voorbeeld: werken met lokale variabelen

Operators in Transact SQL

- **Arithmetic operators**
  - -, +, *, /, %(modulo)
- **Comparison operators**
  - <, >, =, ... , IS NULL, LIKE, BETWEEN, IN
- **Alphanumeric operators**

  + (string concatenation)
- **Logic operators**
  - AND, OR, NOT
- **Function**
  - Arithmetic : ROUND, POWER, COS, ...
  - Date/time : DATEADD, DATEDIFF, GETDATE, DAY, MONTH,...
  - Alphanumeric : LEFT, RIGHT, LTRIM, RTRIM, TRIM, REPLACE, UPPER, LOWER,...
  - System functions : CAST, CONVERT, ISNUMERIC, ISDATE, PRINT,...

control flow with Transact SQL

- Program flow: if/else, while, begin/end

```
create procedure ShowFirstXEmps @x int output,@missed int output
as
declare @empid int, @name varchar(100), @city nvarchar(30), @total int

set @empid = 1
select @total = count(*) from employee
set @missed = 0

if @x > @total
 select @x = count(*) from employee
else
 set @missed = @total - @x

while @empid <= @x
begin
 select @name = lastname + ' ' + firstname,@city= city from employee
 where employeeid = @empid
 print 'Name : ' + @name
 print '   City : '+ @city
 print '----------------------’
 set @empid = @empid + 1
end
```

Comments

- Inline comments
  - -- comment
- Block comments
  - /* comment */

```
create procedure ShowXEmps @x int output,@missed int output
as
/* this procedure retrieves the first x employees,
   assuming employeeid's are numbered 1,2,3,4, ...*/
declare @empid int
declare @name varchar(100)
declare @city nvarchar(30)
```

creation of SP

```
CREATE PROCEDURE <proc_name> [parameter declaratie]
AS
<sql_statements>
```

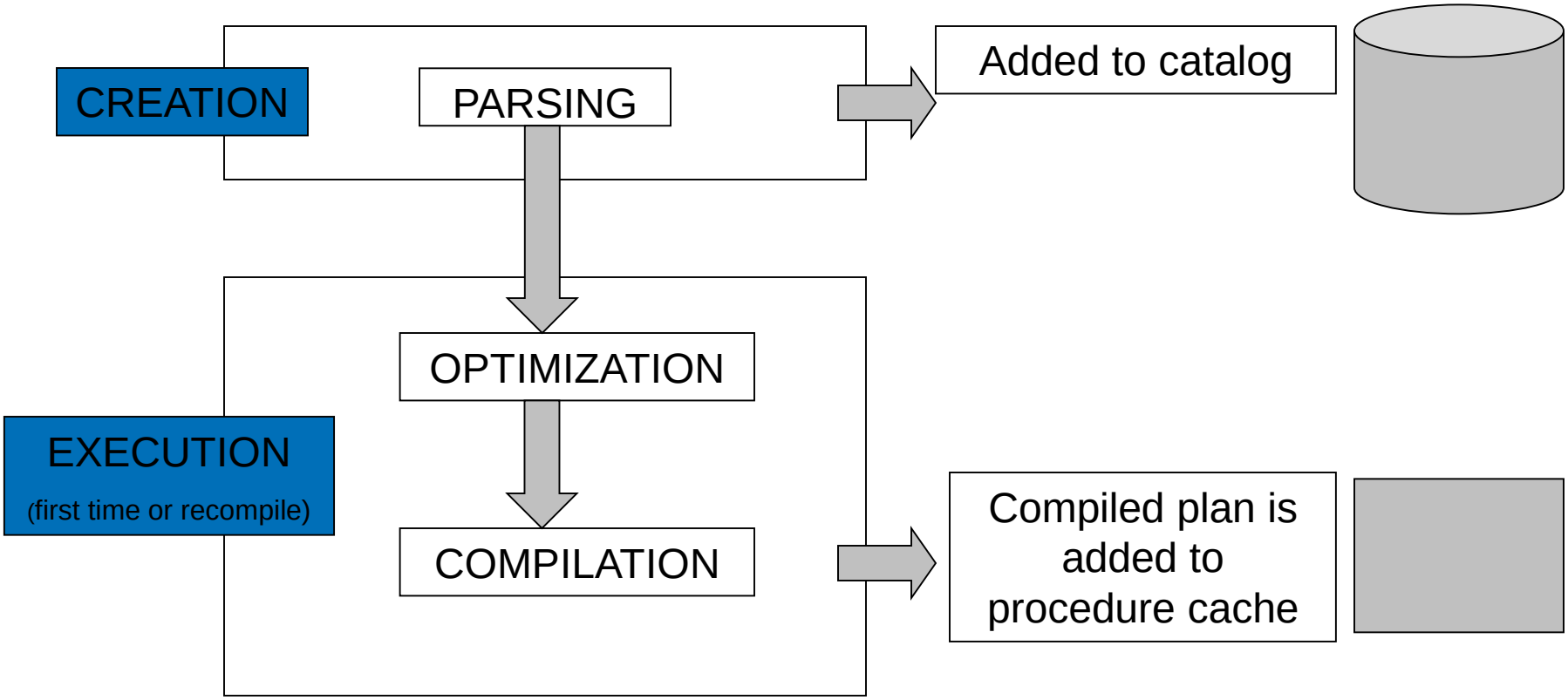
- create db object: via DDL instruction
- Syntax control upon creation
  - The SP is only stored in de DB if it is syntactically correct

```
CREATE PROCEDURE OrdersSelectAll
AS
select * from orders
```

Creation of a Stored procedure without parameters

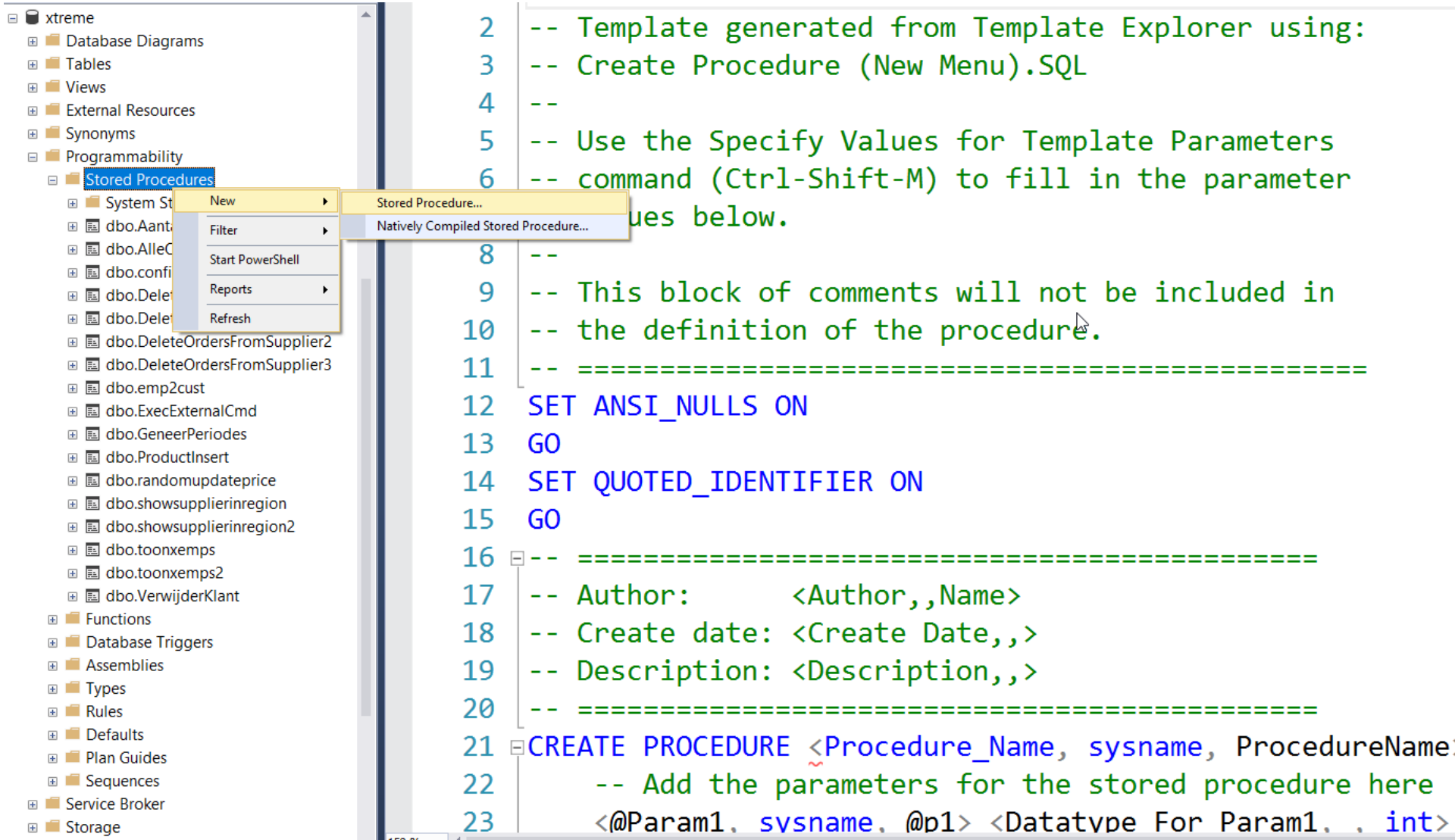
Procedural database objects

– Initial behaviour of a stored procedure



creation of SP

- via SQL Server Management Studio



Changing and removing a SP

```
ALTER PROCEDURE <proc_name> [parameter declaratie]
AS
<sql_statements>
```

```
ALTER  PROCEDURE OrdersSelectAll @customerID int
AS
SELECT * FROM orders
WHERE customerID = @customerID
```

Changing a stored procedure

```
DROP PROCEDURE <proc_name>
```

```
DROP PROCEDURE OrdersSelectAll
```

Removing a stored procedure

Execution of SP

```
EXCUTE <proc_name> [parameters]
```

```
EXECUTE OrdersSelectAll 5
EXEC OrdersSelectAll 5
```

- At first execution
  - compilation and optimisation
- Force recompilation
  - Useful when change of database structure
    - `execute OrdersSelectAll with recompile`
    - `execute sp_recompile OrdersSelectAll`

return value of SP

- After execution a SP **returns a value**
  - Of typ **int**
  - Default return value = 0
- RETURN statement
  - Execution of SP stops
  - Return value can be specified

```
CREATE PROCEDURE OrdersSelectAll AS
  select * from orders
  return @@ROWCOUNT
```

Creation of SP with explicit return value

```
DECLARE @nrOfOrders int
EXEC @nrOfOrders = OrdersSelectAll
PRINT 'We have ' + str(@nrOfOrders) + ' orders.'
```


Use of SP with return value.
Result of print comes in Messages tab.

SP with parameters

- Types of parameters
  - A parameter is passed to the SP with an **input** parameter
  - With an **output** you can possibly pass a value to the SP and get a value back

```
CREATE  PROCEDURE OrdersSelectAllForCustomer
  @customerID int,@count int OUTPUT
AS
SELECT @count = count(*)
FROM orders
WHERE customerID = @customerID
```

```
ALTER  PROCEDURE OrdersSelectAllForCustomer
  @customerID int = 5,@count int OUTPUT
AS
SELECT @count = count(*)
FROM orders
WHERE customerID = @customerID
```

specification of default value

SP with 1 input and 1 output param

SP with parameters

- Calling the SP
  - Always provide keyword OUTPUT for output parameters
  - 2 ways to pass actual parameters
    - use formal parameter name *(order unimportant)*
    - positional

```
DECLARE @number int
EXECUTE OrdersSelectAllForCustomer
  @customerID = 5,
  @count = @number OUTPUT
PRINT @number
```

Pass param by explicit use of formal parameters

```
DECLARE @number int
EXEC OrdersSelectAllForCustomer 5, @number OUTPUT
PRINT @number
```

Positional parameter passing

error handling

- **@@error** is a system function that returns the error number of the last executed SQL statement
  - Value 0 = successful execution

```
CREATE PROCEDURE ProductInsert
  @productID int,
  @productName nvarchar(50),
  @productTypeID int
AS
DECLARE @errormsg int
INSERT INTO product(productID,productName,ProductTypeID)
VALUES (@productID,@productName,@productTypeID)

-- save @@error to avoid overwriting by consecutive statements
SET @errormsg = @@error

IF @errormsg = 0
        PRINT 'SUCCESS! The product has been added.'
ELSE IF @errormsg = 515
        PRINT 'ERROR! ProductID or productName is NULL.'
ELSE IF @errormsg = 547
        PRINT 'ERROR! productTypeID doesn''t exist.'
ELSE PRINT 'ERROR! Unable to add new product. Error:' + str(@errormsg)

RETURN @errormsg
```

Use of @@error to generate customised error messages

Error handling in Transact SQL

- **RETURN**
  - Immediate execution of the batch procedure
- **@@error**
  - Contains error number of last executed SQL instruction
  - Value = 0 if OK
- **RAISERROR**
  - Returns user defined error or system error

- **Use of TRY …. CATCH block**

error handling

- Alle system error message are in the system table **sysmessages**
  - `SELECT * FROM master.dbo.sysmessages WHERE error = @@ERROR`
- Create own messages using **raiserror**
  - raiserror(msg, severity, state)
    - msg – the error message
    - severity – value between 0 and 18
    - state – value between 1 and 127, to distinguish between consecutive calls with same message.
    - Example: replace first ELSE branch in procedure ProductInsert by:

```
ELSE IF @errormsg = 515
BEGIN
    RAISERROR ('ProductID or productName is NULL.',18,1)
END
```

- Example of other system function: **@@rowcount**
  - Count of affected row by last SQL instruction

Exception handling

```
CREATE PROCEDURE TestError (@e int OUTPUT)
AS
BEGIN
   SET @e = 0;
   BEGIN TRY
      INSERT INTO Courier (CourierID,CourierName) VALUES (1,'Test');
   END TRY
   BEGIN CATCH
      SET @e = ERROR_NUMBER();
      PRINT N'Error Procedure = ' + ERROR_PROCEDURE();
      PRINT N'Error Message = ' + ERROR_MESSAGE();
   END CATCH
END

GO
DECLARE @e int;
EXEC TestError @e OUTPUT;
PRINT N'Error code = ' + CAST(@e AS nvarchar(10));
```

Exceptions: catch-block functions

- ERROR_LINE(): line number where exception occurred
- ERROR_MESSAGE(): error message
- ERROR_PROCEDURE: SP where exception occurred
- ERROR_NUMBER(): error number
- ERROR_SEVERITY(): severity level

Exceptions: real life example

```
ALTER procedure UndoLastInvoice @invoicenr char(7) out
as
begin
    BEGIN TRY
        begin transaction
        declare @projectcode int
        select @invoicenr=max(nr) from invoice where nr not like '%A%' AND nr
        not like '%C%'
        select @projectcode=projectcode from invoice where nr=@invoicenr
        delete from invoice where nr=@invoicenr
        update project set invoicedate = null,toinvoice=1 where
        code=@projectcode
        commit
    END TRY
    BEGIN CATCH
        rollback
        insert into log
        values(getdate(),error_message(),error_number(),error_procedure(),
        error_line(),error_severity())
    END CATCH
end
```

Throw

- Is an alternative to RAISERROR
- Without parameters: only in catch block (= rethrowing)
- With parameters: also outside catch block

Throw: without parameters

```
begin try
  insert into courier
  (courierid,CourierName) values
  (1,'test');
end try
begin catch
  print 'This is an error';
  throw
end catch
```

Throw: with parameters

```
throw 52000, 'This is also an error', 1
begin
try
  insert into courier (courierid,CourierName)
  values (1,'test');
end try
begin catch
  print 'This is an error';
  throw
end catch
```

SP example: DeleteCustomer revisited

```
create procedure DeleteCustomer @custno int = NULL
as

if @custno IS NULL
begin
    print 'Please provide a customerid'
    return
end

if not exists (select NULL from customer where customerid=@custno)
begin
    print 'The customer doesn''t exist.'
    return
end

if exists (select NULL from orders where customerid=@custno)
begin
    print 'The customer already has orders and can''t be deleted.'
    return
end

delete from customer where customerid = @custno
print 'The customer has been succesfully deleted'
```


Remove a customer but check if allowed

SP example: Insert Customer

```
CREATE procedure CustomerInsert
  @customerid nchar(5),
  @customername nvarchar(40),
  @address nvarchar(60) = NULL,
  @city nvarchar(15) = NULL,
  @region nvarchar(15) = NULL,
  @postalcode nvarchar(10) = NULL,
  @country nvarchar(15) = NULL
AS
INSERT INTO customer (customerID, customername, address, city,
region, postalcode, country)
VALUES (@customerID, @customername, @address, @city, @region,
@postalcode, @country)
```

```
exec CustomerInsert 1000,'University College Ghent'
```

Add customer

SP example: InsertCustomer with identity

```
ALTER procedure CustomerInsert
  @customername nvarchar(40),
  @address nvarchar(60) = NULL,
  @city nvarchar(15) = NULL,
  @region nvarchar(15) = NULL,
  @postalcode nvarchar(10) = NULL,
  @country nvarchar(15) = NULL,
  @customerid nchar(5) OUTPUT
AS
INSERT INTO customer (customername, address, city,
  region, postalcode, country)
VALUES ( @customername, @address, @city, @region,
  @postalcode, @country)
SET @customerid = @@identity
```

Add customer but now customerID is generated automatically as an identity. Return the generated customer ID to the calling environment

Functions

- standard SQL functions:
  min,max,sum,avg,count

- non-standard built-in functions:

  SQL Server: datediff, substring, len, round, …

  http://technet.microsoft.com/en-us/library/ms174318.aspx

- user defined functions

Why user defined functions?

- Give the age of each employee:

```
select lastname,firstname,cast(birthdate as date) birthdate,
cast(getdate() as date) today,datediff(year,birthdate,getdate()) age
from employee;
```

lastname	firstname	birthdate	today	age
Davolio	Nancy	1988-12-08	2019-03-20	31
Fuller	Andrew	1985-02-19	2019-03-20	34
Leverling	Janet	1987-08-30	2019-03-20	32
Peacock	Margaret	1989-09-19	2019-03-20	30
Buchanan	Steven	1995-09-11	2019-03-20	24

Will probably not be happy

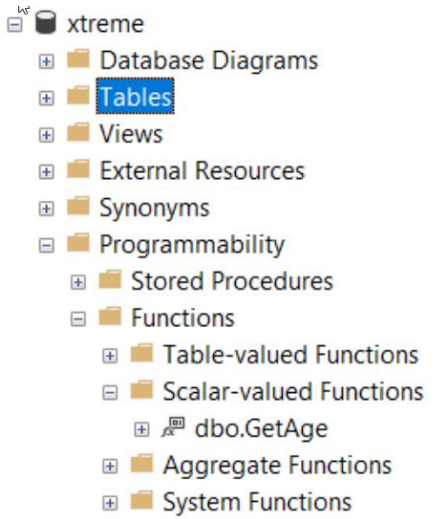
```
select lastname,firstname,cast(birthdate as date) birthdate,
cast(getdate() as date) today,datediff(day,birthdate,getdate())/365 age
from employee;
```

lastname	firstname	birthdate	today	age
Davolio	Nancy	1988-12-08	2019-03-20	30
Fuller	Andrew	1985-02-19	2019-03-20	34
Leverling	Janet	1987-08-30	2019-03-20	31
Peacock	Margaret	1989-09-19	2019-03-20	29
Buchanan	Steven	1995-09-11	2019-03-20	23

Better but doesn't take into account leap years

Solution: User Defined Fuction

```
CREATE FUNCTION GetAge
    (@birthdate AS DATE,@eventdate AS DATE)
RETURNS INT
AS
BEGIN
 RETURN
    DATEDIFF(year, @birthdate, @eventdate)
    - CASE WHEN 100 * MONTH(@eventdate) + DAY(@eventdate) <
                100 * MONTH(@birthdate) + DAY(@birthdate)
    THEN 1 ELSE 0
 END;
END;
```



User Defined Functions

How to use:

```
select lastname,firstname,cast(birthdate as date) birthdate,
cast(getdate() as date) today,
dbo.GetAge(birthdate,getdate()) age
from employee;
```

lastname	firstname	birthdate	today	age
Davolio	Nancy	1988-12-08	2019-03-20	30
Fuller	Andrew	1985-02-19	2019-03-20	34
Leverling	Janet	1987-08-30	2019-03-20	31
Peacock	Margaret	1989-09-19	2019-03-20	29
Buchanan	Steven	1995-09-11	2019-03-20	23

User Defined Functions

- (db xtreme): give per product class the price of the cheapest product that costs more than *x* € an a product with that price.
- You can solve this using a inline table valued function

```
create function minimum (@limit int) returns table
as
return
select productclassid class,min(price) minprice
from product p where price >= @limit
group by productclassid;
-- use:
select class,minprice,
(select top 1 productid from product where
productclassid=class and price=minprice)
from minimum(0);
```

Besides views and CTE's, this is another way to reuse SELECT statements, now even parameterised!

Exercises

- DB xTreme

  Write a stored procedure to delete all orders for a given supplier. Return the number of deleted orders as an output parameter.

  Test your stored procedure.

  Why can’t you delete the orders?

  Solution: see next section about “Cursors”

Exercise

DB xTreme

Create a stored procedure for deleting a product. You can only delete a product if

1. The product exists
2. There are no purchases for the product
3. There are no orders for the product

Write two versions of your procedure:

a) In the first version you check these conditions before deleting the product, so you don't rely on SQL Server messages. Generate an appropriate error message if the product can't be deleted.

b) In the second version you try to delete the product and catch the exeptions that might occur.

Test your procedure. Give the select's to find appropriate test data.

CURSORS

CURSORS

SQL statements are processing **complete resultsets** and not individual rows. Cursors allow to process **individual rows** to perform complex row specific operations that can't (easily) be performed with a single SELECT, UPDATE or DELETE statement.

- A cursor is a database object that refers to the result of a query. It allows to specify the row from the resultset you wish to process.

- 5 important cursor related statements
  - **DECLARE CURSOR** – creates and defines the cursor
  - **OPEN** - opens the declared cursor
  - **FETCH** - fetches 1 row
  - **CLOSE** – closes the cursor (counterpart of OPEN)
  - **DEALLOCATE** – remove the cursor definition (counterpart of DECLARE)

Cursor declaration

```
DECLARE <cursor_name> [INSENSITIVE][SCROLL] CURSOR FOR
<SELECT_statement>
[FOR {READ ONLY | UPDATE[OF <column list>]}]
```

- **INSENSITIVE**
  - The cursor uses a **temporary copy** of the data
    - Changes in underlying tables after opening the cursor are not reflected in data fetched by the cursor
    - The cursor can't be used to change data (read-only, see below)
  - if “insensitive” is omitted, deletes and updates are reflected in the cursor

    → less performant because each row fetch executes a SELECT
- **SCROLL**
  - All fetch operations are allowed
    - FIRST, LAST, PRIOR, NEXT, RELATIVE and ABSOLUTE
    - Might result in difficult to understand code
  - If “scroll” is omitted only NEXT can be used

Cursor declaration *_continued*

```
DECLARE <cursor_name> [INSENSITIVE][SCROLL] CURSOR FOR
<SELECT_statement>
[FOR {READ ONLY | UPDATE[OF <column list>]}]
```

- **READ ONLY**
  - Prohibits data changes in underlying tables through cursor
- **UPDATE**
  - Data changes are allowed
  - Specify columns that can be changed via the cursor

```
DECLARE orders_cursor CURSOR FOR
select OrderID,OrderAmount,customername from orders o join customer c
on o.CustomerID= c.customerid
where OrderAmount > 10000;
```

Example of a cursor declaration

Opening a cursor

```
OPEN <cursor name>
```

- The cursor is opened
- The cursor is "filled"
  - The select statement is executed. A "virtual table" is filled with the "active set".
- The cursor's **current row pointer** is positioned just before the first row in the result set.

```
OPEN orders_cursor
```

Fetching data with a cursor

```
FETCH [NEXT | PRIOR | FIRST | LAST | {ABSOLUTE | RELATIVE
<row number>}]
FROM <cursor name>
[INTO <variable name>[,...<last variable name>]]
```

- The cursor is positioned
  - On the next (or previous, first, last, …) row
  - Default only NEXT is allowed
    - For other ways use aSCROLL-able cursor
- Data is fetched
  - without INTO clause resulting data is shown on screen
  - with INTO clause data is assigned to the specified variables
    - Declare a corresponding variable for each column in the cursor SELECT.

Fetching data with a cursor _continued

```
DECLARE @orderid int,@orderamount decimal(18,2),@customername nvarchar(40)

FETCH NEXT FROM orders_cursor INTO @orderid, @orderamount, @customername

WHILE @@FETCH_STATUS = 0
BEGIN
  PRINT 'Order: ' + str(@orderid) + ', Amount:  '
        +  str(@orderamount) + ', Customer: ' + @customername
  FETCH NEXT FROM orders_cursor INTO @orderid, @orderamount, @customername
END
```

Example of data fetch

Closing a cursor

```
CLOSE <cursor name>
```

- The cursor is closed
  - The cursor definition remains
    - Cursor can be reopend

```
CLOSE orders_cursor
```

Deallocating a cursor

```
DEALLOCATE <cursor name>
```

- The cursor definition is removed
  - If thas was the last reference to the cursor all cursor resources (the "virtual table") are released.
  - If the cursor has not been closed yet DEALLOCATE will close the cursor

```
DEALLOCATE orders_cursor
```

Overview CURSOR example

```
DECLARE @orderid int,@orderamount decimal(18,2),@customername nvarchar(40)

DECLARE orders_cursor CURSOR FOR
select OrderID,OrderAmount,customername from orders o join customer c
on o.CustomerID= c.customerid
where OrderAmount > 10000;

OPEN orders_cursor

FETCH NEXT FROM orders_cursor INTO @orderid, @orderamount, @customername

WHILE @@FETCH_STATUS = 0
BEGIN
  PRINT 'Order: ' + str(@orderid) + ', Amount:  '
        +  str(@orderamount) + ', Customer: ' + @customername
  FETCH NEXT FROM orders_cursor INTO @orderid, @orderamount, @customername
END

CLOSE orders_cursor

DEALLOCATE orders_cursor
```

Nested cursors: details per order

```
DECLARE @orderid int,@orderamount decimal(18,2),@customername nvarchar(40)

DECLARE @productid int,@quantity int

DECLARE orders_cursor CURSOR FOR
  select OrderID,OrderAmount,customername from orders o join customer c
  on o.CustomerID= c.customerid
  where OrderAmount > 10000;

OPEN orders_cursor

FETCH NEXT FROM orders_cursor
INTO @orderid, @orderamount, @customername

WHILE @@FETCH_STATUS = 0
BEGIN
  PRINT 'Order: ' + str(@orderid) + ', Amount:  ' +  str(@orderamount) + ', Customer: ' + @customername
  --  begin inner cursor
  DECLARE details_cursor CURSOR FOR select productid,quantity from OrdersDetail where OrderID=@orderid
  OPEN details_cursor
  FETCH NEXT FROM details_cursor INTO @productid, @quantity
  WHILE @@FETCH_STATUS = 0
  BEGIN
    PRINT '      ' + 'Product: ' + str(@productid) + ', Quantity:' + str(@quantity)
    FETCH NEXT FROM details_cursor INTO @productid, @quantity
  END
  CLOSE details_cursor
  DEALLOCATE details_cursor
  -- end inner cursor
  FETCH NEXT FROM orders_cursor INTO @orderid, @orderamount, @customername
END

CLOSE orders_cursor

DEALLOCATE orders_cursor
```

update and delete via cursors

```
DELETE FROM <table name>
WHERE CURRENT OF <cursor name>
```

```
UPDATE <table name>
SET ...
WHERE CURRENT OF <cursor name>
```

- positioned update/delete
  - Deletes/updates the row the cursor referred in **WHERE CURRENT OF** points to
  - = cursor based positioned update/delete
  - Example (see inner cursor on previous slide): delete all orderdetails with quantity < 5:

```
WHILE @@FETCH_STATUS = 0
 BEGIN
   PRINT '      ' + 'Product: ' + str(@productid) + ', Quantity:' + str(@quantity)
   IF @quantity < 5
      DELETE FROM ordersdetail WHERE CURRENT OF details_cursor
   FETCH NEXT FROM details_cursor INTO @productid, @quantity
 END
```

Exercise

- Go back to the exercise to delete all orders that contain products from a given supplier.
- Now delete the orders (and orderdetails) using a cursor.
- Also show the number of deleted orders and the number of deleted orderdetails

triggers

Triggers

A trigger: a database program, consisting of procedural and declarative instructons, saved in the catalogue and activated by the DBMS if a certain operation on the database is executed and if a certain condition is satisfied.

- Comparable to SP but can't be called explicitly
  - A trigger is automatically called by the DBMS with some DML, DDL, LOGON-LOGOFF commands
    - DML trigger: with **an insert, update or delete** for a table or view *(in this course we further elaborate this type of cursors)*
    - DDL trigger: executed with a **create, alter of drop** of a table
    - LOGON-LOGOFF trigger: executed when a user logs in or out
      (MS SQL Sever: only LOGON triggers, Oracle: both)

Triggers

- DML triggers
  - Can be executed with
    - **insert**
    - **update**
    - **delete**
  - Are activated
    - **before** – before the IUD is processed *(not supported by SQL Server)*
    - **instead of** - instead of IUD command
    - **after** – after the IUID is processed (but before COMMIT), this is the default
  - *In some DMBS (ex. Oracle) you can also specify how many times the cursor is activated*
    - *for each row*
    - *for each statement*

Procedural database objects

- procedural programs

types	Saved as	execution	Supports parameters
script	Separate file	client tool (ex. Management Studio)	no
stored procedure	database object	via application or SQL script	yes
user defined function	database object	via applicaton or SQL script	yes
trigger	database object	via DML statement	no

Why using triggers ?

- Validation of data and complex constraints
  - An employee can't be assigned to > 10 projects
  - An employee can only be assigned to a project that is assigned to his department
- Automatic generation of values
  - If an employee is assigned to a project the default value for the monthly bonus is set according to the project priority and his job category
- Support for alerts
  - Send automatic e-mail if user if an employee is removed from a project
- Auditing
  - Keep track of who did what on a certain table
- Replication and controlled update of redundant data
  - Db xtreme: If an ordersdetail record changes update the orderamount in the orders table
  - Automatic update of datawarehouse tables for reporting (see chapter "Datawarehousing")

Advantages and disadvantages

- Major advantage
  - Store functionality in the DB and execute consistently with each change of data in the DB
- Consequences
  - no redundant code
    - functionality is localised in a single spot, not scattered over different applications (desktop, web, mobile), written by different authors
  - written & tested 'once' by an experienced DBA
  - security
    - triggers are in the DB so all security rules apply
  - more processing power
    - for DBMS and DB
  - fits into client-server model
    - 1 call to db-serve: al lot can happen without further communication

Advantages and disadvantages

- Drawbacks
  - complexity
    - DB design, implementation are more complex by shifting functionality from application to DB
    - Very difficult to debug
  - Hidden functionality
    - The user can be confronted with unexpected side effects from the trigger, possibly unwanted
    - Triggers can cascade, which is not always easy to predict when designing the trigger
  - Performance
    - At each database change the triggers have to be reevaluated
  - Portability
    - Restricted to the chosen database dialect (ex. Transact-SQL from MS)

Comparison of trigger functionality

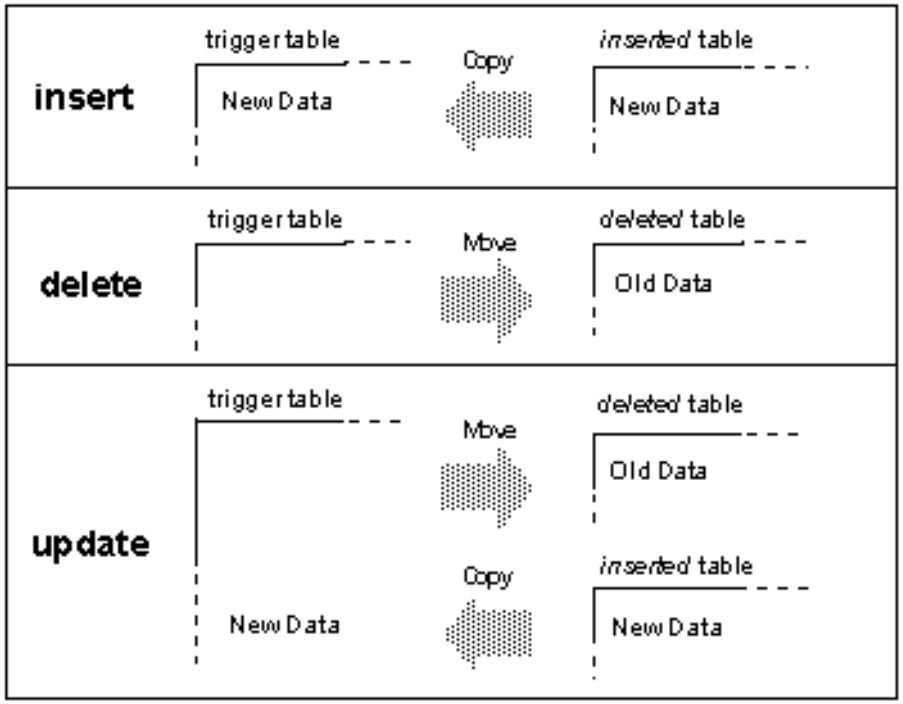
	Oracle	MS SQL Server	MySQL
BEFORE *For validation*	X	simulate via AFTER-trigger+ROLLBACK	X
AFTER	X	X	X
INSTEAD OF *for views*	X	X	N/A
FOR EACH STATEMENT	X	default	default
FOR EACH ROW	X Acces to values before/after per row via :NEW/:OLD vars	N/A Acces to values before/after per row via deleted/inserted pseudo-tables and cursors	X Acces to values before/after per row via NEW/OLD vars
TRANSACTIONS	COMMIT/ROLLBACK Not allowed	COMMIT/ROLLBACK Allowed	COMMIT/ROLLBACK Not allowed

"Virtual" tables with triggers

- 2 temporary tables
  - ***deleted*** table
    contains copies of updated and deleted rows
    - During update or delete **rows are moved from the triggering table to the *deleted* table**
    - Those two table have no rows in common
  - ***inserted*** table
    contains copies of updated or inserted rows
    - During update or insert **each affected row is copied from the triggering table to the *inserted* table**
    - All rows from the *inserted* table are also in the triggering table

trigger virtual tables

- **deleted** and **inserted** table



creation of an after trigger

- Only by SysAdmin or dbo
- Linked to **one table**; **not to a view**

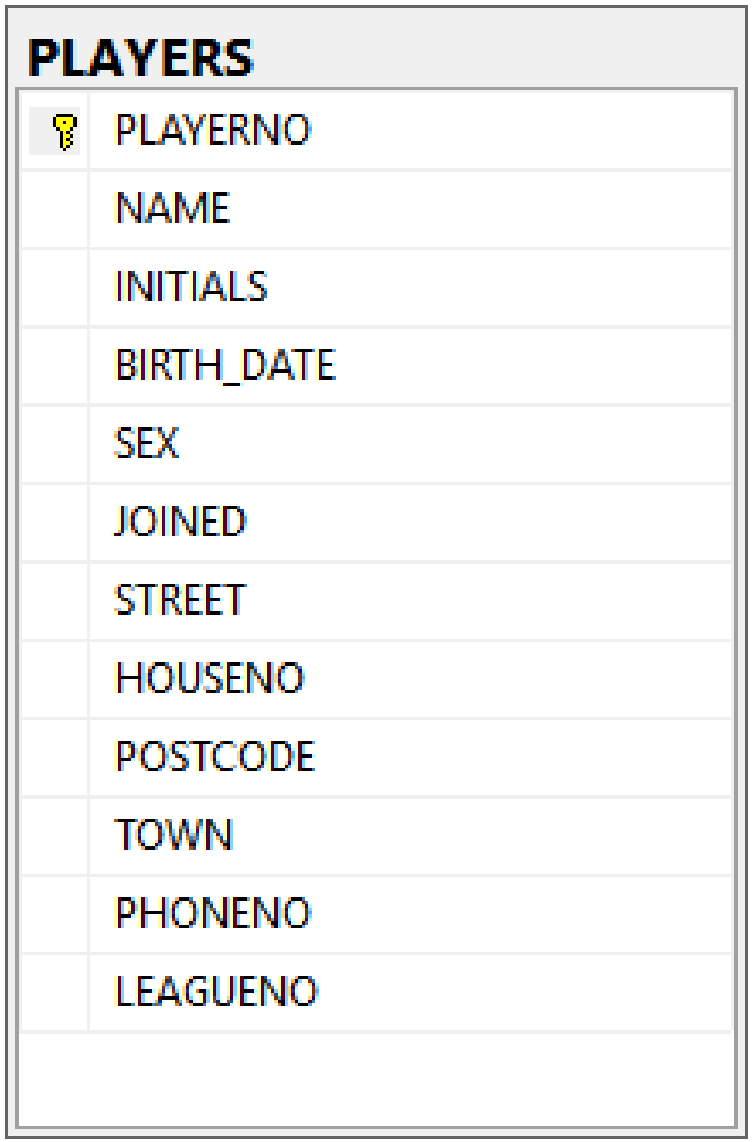
- Is executed
  - After execution of the triggering action, i.e. *insert, update, delete*
  - After copy of the changes to the temporary tables *inserted* and *deleted*
  - Before COMMIT

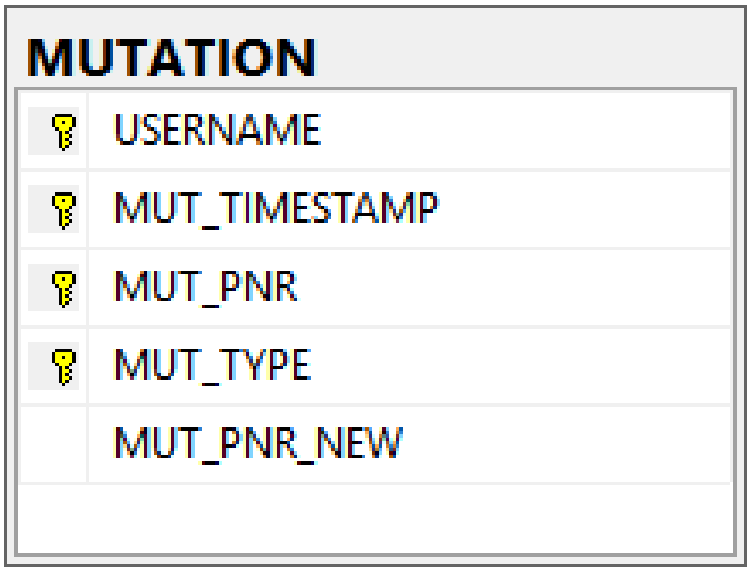
```
CREATE TRIGGER naam
ON tabel
FOR [INSERT, UPDATE, DELETE]
AS ...
```

vereenvoudigde syntax voor een after trigger

Creation of trigger

- Example (db Tennis)





insert after-trigger

- triggering instruction is an **insert** statement
  - **inserted** – logical table with columns equal to columns of triggering table, containing a copy of inserted rows
  - Remark: when triggering by INSERT-SELECT statement more than one record can be added at once. The trigger code is executed only once, but will insert a mutation record for each inserted record

```
CREATE TRIGGER insert_player ON PLAYERS FOR insert
AS
 INSERT INTO mutation
(username, mut_timestamp, mut_pnr, mut_type, mut_pnr_new)
 SELECT user, getdate(), null, 'i', playerno FROM inserted
```

Automatic insert in mutation table when adding players

delete after-trigger

- triggering instruction is a **delete** instruction
  - **deleted** - logical table whith columns equal to columns of triggering table, containing a copy of delete rows
  - We use a stored procedure that we can reuse in the update trigger later on.

```
CREATE PROCEDURE usp_mutation_insert
  (@MPNR SMALLINT,
   @MTYPE CHAR(1),
   @MPNR_NEW SMALLINT)
AS
  INSERT INTO mutation
  (username, mut_timestamp, mut_pnr,mut_type, mut_pnr_new)
  VALUES (user, getdate() ,@MPNR, @MTYPE, @MPNR_NEW);
```

delete after-trigger

Automatic insert in mutation table
*upon deleting one **or several** players*

```
CREATE TRIGGER  delete_player
   ON players FOR delete
AS
    DECLARE @old_pnr smallint
    DECLARE del_cursor CURSOR FOR SELECT playerno FROM deleted
    OPEN del_cursor
    FETCH NEXT FROM del_cursor INTO @old_pnr
    WHILE @@FETCH_STATUS = 0
    BEGIN
        EXEC usp_mutation_insert @old_pnr,'D',null
        FETCH NEXT FROM del_cursor INTO @old_pnr
    END
    CLOSE del_cursor
    DEALLOCATE del_cursor
```

```
Activation of the trigger:
 delete from players where playerno > 115;
```

update after-trigger

- triggering instruction is an **update** instruction

```
CREATE TRIGGER  update_player ON players FOR update
AS
DECLARE @old_pnr smallint, @new_pnr smallint

DECLARE before_cursor CURSOR FOR SELECT playerno FROM deleted
ORDER BY playerno

DECLARE after_cursor  CURSOR FOR SELECT playerno FROM inserted
ORDER BY playerno

OPEN before_cursor
OPEN after_cursor

FETCH NEXT FROM before_cursor INTO @old_pnr
FETCH NEXT FROM after_cursor  INTO @new_pnr

WHILE @@FETCH_STATUS = 0
BEGIN
     EXEC usp_mutation_insert @old_pnr,'U',@new_pnr
     FETCH NEXT FROM before_cursor INTO @old_pnr
     FETCH NEXT FROM after_cursor  INTO @new_pnr
END

DEALLOCATE before_cursor
DEALLOCATE after_cursor
```

update after-trigger

- Activation of the trigger:

```
 update players set joined = joined + 20;

OR:

 update players set playerno = playerno + 100;
```

the IF UPDATE clause

- Conditional execution of triggers: execute only if a specific columns is mentioned in update or insert

```
ALTER TRIGGER  update_player ON players FOR update
AS
DECLARE @old_pnr smallint, @new_pnr smallint

DECLARE before_cursor CURSOR FOR SELECT playerno FROM deleted  ORDER BY playerno
OPEN before_cursor
IF update(playerno)
BEGIN
  DECLARE after_cursor  CURSOR FOR SELECT playerno FROM inserted ORDER BY playerno
  OPEN after_cursor
END

FETCH NEXT FROM before_cursor INTO @old_pnr
IF update(playerno)
  FETCH NEXT FROM after_cursor  INTO @new_pnr
ELSE
  SET @new_pnr = @old_pnr

WHILE @@FETCH_STATUS = 0
BEGIN
  EXEC usp_mutation_insert @old_pnr,'U',@new_pnr
  FETCH NEXT FROM before_cursor INTO @old_pnr
  IF update(playerno)
    FETCH NEXT FROM after_cursor  INTO @new_pnr
  ELSE
    SET @new_pnr = @old_pnr
END

DEALLOCATE before_cursor
IF update(playerno)
  DEALLOCATE after_cursor
```

other trigger conditions

- "normal" conditions are also possible

```
IF datepart(hour, getdate()) >= 9
AND datepart(hour, getdate()) < 19

    BEGIN ... END
```

Only execute between 9:00 and 19:00

```
IF  USER IN  ('JAN', 'PETER', 'MARK')
     BEGIN...END
```

Triggercode is only executed for specific users…

Example: controlled update of redundant data

- Suppose the players table has a field SUM_PENALTIES ***(redundantce)***. This field stores for each player the sum of his/her penalties. Now we want to create triggers that automatically update this field ***(integrity)*** .

```
CREATE TRIGGER penalty_insert ON penalties
FOR INSERT
AS
  DECLARE @pen smallint, @pnr smallint
  SELECT @pen = amount, @pnr = playerno from inserted
  update players set sum_penalties = sum_penalties + @pen
  WHERE playerno = @pnr
```

- Remark: this trigger only works if inserts always happen one-by-one (because of `SELECT @pen = amount, @pnr = playerno from inserted`)
- If this cannot be guaranteed (because of e.g. INSERT SELECT statements), then use a cursor (see examples above).

Example: controlled update of redundant data

- Possible approach for update and delete triggers

```
CREATE TRIGGER pen_del_upd ON penalties
FOR UPDATE, DELETE
AS
    DECLARE @pen as smallint, @pnr as smallint
    SELECT @pnr = playerno from deleted
    SELECT @pen = SUM(amount) FROM penalties WHERE playerno = @pnr
    UPDATE players SET sum_penalties = @pen
    WHERE playerno = @pnr
```

- Remark: this trigger only works if updates and deletes are guaranteed to happen on-by-one
  (because of `SELECT @pnr = playerno from deleted`)
- Can this trigger also be used for insert?

Triggers and transactions

- a trigger is part of the same transaction as the triggering instruction
- inside the trigger this transaction can be ROLLBACKed
- ex. a player who is team coach can never be deleted (suppose there are no foreign key constraints)

→ although a trigger in SQL Server occurs after the triggering instruction, that instruction can still be undone in the trigger

```
CREATE TRIGGER delplayer ON PLAYERS
FOR delete
AS
IF (SELECT COUNT(*)
    FROM deleted JOIN teams
    ON teams.playerno = deleted.playerno) > 0
BEGIN
  ROLLBACK TRANSACTION
  RAISERROR ('The player is team coach.', 14,1)
END
```

Find all tables with triggers

The catalog is a set of sytem tables and views providing meta information about the database. You can query those tables to avoid endless clicking in the Object Explorer. Example: find all tables with at least one trigger.

```
DECLARE @min_count INT;
SET @min_count = 1;

SELECT [table] = s.name + N'.' + t.name
  FROM sys.tables AS t
  INNER JOIN sys.schemas AS s
  ON t.[schema_id] = s.[schema_id]
  WHERE EXISTS
  (
    SELECT 1 FROM sys.triggers AS tr
      WHERE tr.parent_id = t.[object_id]
      GROUP BY tr.parent_id
      HAVING COUNT(*) >= @min_count
  );
```

Tables and User Defined Types

User defined types

- ~abstract data types
- Can be used as built-in types
- 2 sorts
  - distinct types
  - structured types: tables
- Despite the SQL:2008-standard, many differences between DMBS's

Distinct type

- Based on a basic type
- Allows to refine existing data types
- Example (MS-SQL Server)

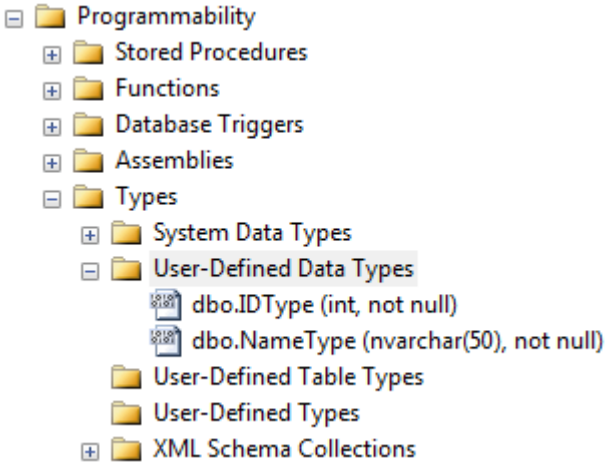
```
CREATE TYPE IDType FROM INT NOT NULL;
CREATE TYPE NameType FROM NVARCHAR(50) NOT NULL;
```

Distinct Type

Basic type of which distinct type inherits

- *The new type is stored in the databse*
- *The distinct type can be used as if it were a built-in type*

Dinstinct Type: use



Remove from the database:

DROP Type IDType;

```
CREATE TABLE PERSON
(id IDType PRIMARY KEY,name NameType);
```

Usage:

- Ensure that all person names in the database use the same type

More info: http://technet.microsoft.com/en-us/library/ms175007.aspx

Table variables

- Local temporary tables: #table
- Global tempory tables: ##table
- Table variables: @table
- Table types

Local temporary tables

- Stored in tempdb under "System Databases"
- Only visible:
  - In the creating session
  - At the creating level
  - In all underlying levels
- Removed if creating level goes out-of-scope

Local temporary tables

```
IF OBJECT_ID('tempdb.dbo.#MyOrderTotalsByYear') IS NOT NULL
DROP TABLE dbo.#MyOrderTotalsByYear;


CREATE TABLE #MyOrderTotalsByYear
( orderyear INT NOT NULL PRIMARY KEY, qty INT NOT NULL );


INSERT INTO #MyOrderTotalsByYear(orderyear, qty)
SELECT  YEAR(O.orderdate) AS orderyear,SUM(OD.quantity) AS qty
FROM Orders AS O JOIN OrdersDetail AS OD ON OD.orderid = O.orderid
GROUP BY YEAR(orderdate);


SELECT Cur.orderyear, Cur.qty AS curyearqty, Prv.qty AS prvyearqty
FROM dbo.#MyOrderTotalsByYear AS Cur
LEFT OUTER JOIN dbo.#MyOrderTotalsByYear AS Prv
ON Cur.orderyear = Prv.orderyear + 1;
```

Global temporary tables

- Stored in tempdb under "System Databases"
- Visible in all sessions
- Removed if creating session disconnects and there or no more references

Global temporary tables

```
CREATE TABLE ##Globals (   id  char(4) NOT NULL PRIMARY KEY, val INT NOT NULL
);

INSERT INTO ##Globals(id, val) VALUES('ABCD', 10);

SELECT val FROM ##Globals WHERE id = 'ABCD';

DROP TABLE ##Globals;
```

Table Variables

```
DECLARE @myordersperyear AS TABLE
(
  year INT NOT NULL PRIMARY KEY,
  quantity INT NOT NULL
);
```

Table Types

```
CREATE TYPE TotalOrdersPerYear AS TABLE
(
 year INT NOT NULL PRIMARY KEY,
 quantity INT NOT NULL
);

DECLARE @myordersperyear AS TotalOrdersPerYear;

INSERT INTO @ myordersperyear
select year(orderdate) as year,round(sum(unitprice*quantity),2)
from orders o join ordersdetail od
on o.OrderID=od.orderid
group by year(orderdate);

SELECT * FROM @ myordersperyear;
```

Table Types and Variables

- table types are stored in the DB
- table variables only exist during batch executions (= sequence of statements)

Advantages of table variables and table types:

- Shorter and cleaner code
- table type variables can also be passed as parameters to stored procedures and functions

Local temporary tables vs. table variables

- Both are only visible in creating session
- Table variables have more limited scope:
  - Only visible in current batch
  - Not visible in other batches in same session

Local temporary tables vs. table variables

```
IF OBJECT_ID('tempdb.dbo.#MyOrderTotalsByYear') IS NOT NULL DROP TABLE dbo.#MyOrderTotalsByYear;
CREATE TABLE #MyOrderTotalsByYear ( orderyear INT NOT NULL PRIMARY KEY, qty INT NOT NULL );

DECLARE @myordersperyear AS TABLE
(
  jaar INT NOT NULL PRIMARY KEY,
  hoeveelheid INT NOT NULL
);

select * from #MyOrderTotalsByYear;
select * from @myordersperyear;

GO  -- ← execute previous commands and start new batch. (same as button “Execute” in SSMS)

select * from #MyOrderTotalsByYear;
select * from @myordersperyear;   -- Must declare the table variable "@myordersperyear "
```

Temporary tables: example

- Due to the Brexit we want to delete all orders that are not yet shipped and that contain products that are supplied by a supplier from the UK.
- We first have to delete the ordersdetail because of the FK constraint with orders
- But after deleting the ordersdetail we loose the link with the supplier and don’t kwow which orders to delete anymore
- Solution: save the orderid’s in a temporary table.

Temporary tables: example

```
create procedure DeleteOrdersUK @deletedorders int output
as
begin
    set nocount on
    create table #OrdersUK (orderid int)

    insert into #OrdersUK
        select distinct od.orderid from ordersdetail od
        join orders o on od.orderid=o.orderid
        where productid in
        (select productid from product p join supplier s on
        p.SupplierID=s.SupplierID where country='UK')
        and o.Shipped=0;

    delete from ordersdetail
        where orderid in (select orderid from #OrdersUK)
    delete from orders
        where orderid in (select orderid from #OrdersUK)
    set @deletedorders = @@rowcount
end
```

Temporary tables: example

```
-- test of procedure DeleteOrdersUK
begin transaction

select od.orderid from ordersdetail od join orders o
on od.orderid=o.orderid
    where productid in
    (select productid from product p join supplier s on
    p.SupplierID=s.SupplierID where country='UK')
    and o.Shipped=0;

declare @nroforders int
exec DeleteOrdersUK @nroforders out
print 'Nr of deletedorders = ' + cast(@nroforders as varchar)

select od.orderid from ordersdetail od join orders o
on od.orderid=o.orderid
    where productid in
    (select productid from product p join supplier s on
    p.SupplierID=s.SupplierID where country='UK')
    and o.Shipped=0;

rollback
```

Temporary tables: exercise

- Go once again back to the exercise to delete all orders that contain products from a given supplier.
- Now use a temporary table instead of a cursor
- Also show the number of deleted orders and the number of deleted orderdetails
- Throw an exception when the given supplier doesn’t exist
- Catch the exception in your test code

Large Objects

Large Objects

- a large object is a datatype that can hold large data
  - Textfile, image, music, video, webpage, xml document, json document
- types
  - BLOB
    - Binary Large OBject
  - CLOB
    - Character Large OBject
  - NCLOB
    - National Character Large OBject

Large Objects

- problems with LOBs used in DBMS's
  - LOBs are considered as byte-stream
  - DBMS can't handle the LOB
    - DBMS does not know content nor internal structure
    - Although often the LOB contains structured data
      - Structured text, web page, …
  - costly transfer of LOB's from server to client over netwerk

Large Objects

- example (MS – SQL Server)

BLOB data types:

- varbinary(max) and image (image = deprecated)

CLOB data types (ASCII data):

- varchar(max)

NCLOB data types (unicode data):

- nvarchar(max)

```
ALTER TABLE Employees
  ADD cv varchar(max);
ALTER TABLE Employees
  ADD foto varbinary(max);
```

DYNAMIC SQL

Early Binding versus Late Binding

- SQL binding refers to the translation of SQL code to a lower-level representation that can be executed by the DBMS, after performing tasks such as validation of table and field names, checking whether the user or client has sufficient access rights, and generating a query plan to access the physical data in the most performant way possible.
- Early versus late binding then refers to the actual moment when this binding step is performed

Early Binding Versus Late Binding

- Early binding is possible in case a pre-compiler is used and can hence only be applied with an embedded API
  - beneficial in terms of performance
  - binding only needs to be performed once
  - pre-compiler can perform specific syntax checks

Early Binding Versus Late Binding

- Late binding performs the binding of SQL-statements at runtime
  - additional flexibility is offered (“dynamic SQL”)
  - syntax errors or authorisation issues will remain hidden until the program is executed
  - testing the application can be harder (use “PRINT”)
  - less efficient for queries that must be executed multiple times
  - Risk of SQL injection attacks (see further)
- Not allowed in UDF’s (= risk of side effects)

Dynamic SQL: example

```
declare @region varchar(10);
set @region = 'OR';
declare @sqlstring varchar(100) = 'select *
from supplier where region=''' + @region +
'''';
exec (@sqlstring);
```

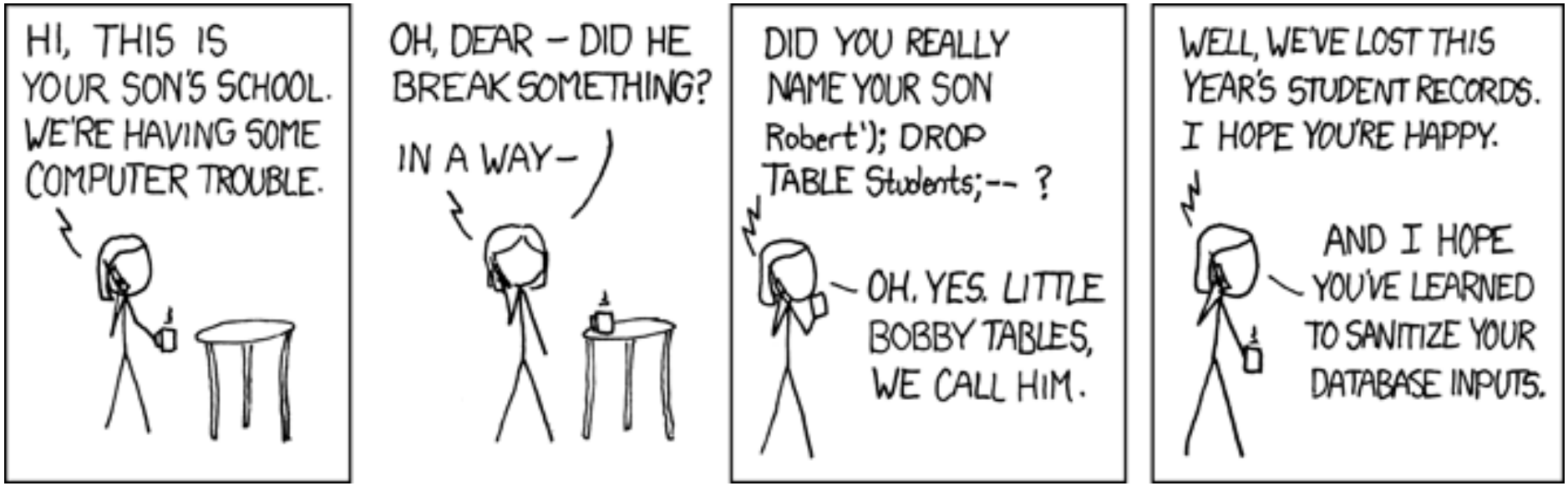
→Disadvantages:

- no cached query execution plan → slower
- debugging is more difficult (use PRINT!)
- Not allowed in UDF's (= risk of side-effect)
- SQL injection

SQL Injection & Dynamic SQL

What if @regio is filled from a GUI like this (by a hacker)?

```
declare @region varchar(50);
set @region = 'OR'';DROP TABLE PRODUCT2;--';
declare @sqlstring varchar(100) = 'select * from supplier where region=''' + @region + '''';
print @sqlstring;
exec (@sqlstring);
```


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

SQL Injection & Dynamic SQL Guidelines

- Never trust front-end data. Check for valid input (ex. only A-Z and 0-9 are allowed)
- Don't allow symbols like `'` `;` `(` `)` `--`
- Pay attention if input data contains sp_ or xp_ → these system stored procedures can be destructive

Dynamic SQL: sp_executesql

- Separate parameters:

```
declare @sqlstring nvarchar(100) =
'select * from supplier where
region=@region';
print @sqlstring;
exec sp_executesql @sqlstring,
'@regio varchar(50)', @region='OR';
```

→ Guard against SQL injection

Dynamic SQL: scope

- Dynamic SQL is executed in its "own" batch:
  - Variables and temporary tables are created in a dynamic SQL statement and are not available in the calling procedure

```
declare @sqlstring nvarchar(200) =
'CREATE TABLE #tempprod(productid int not null primary key,
                        name varchar(200));'
set @sqlstring += 'INSERT INTO #tempprod SELECT
                   productid,productname from product'
execute(@sqlstring);
select * from #tempprod;
```

```
(116 row(s) affected)
Msg 208, Level 16, State 0, Line 6
Invalid object name '#tempprod'.
```

Exercises

- Bundle xTreme
  - 93
- Make sure that countries are not hard coded in this statement (that produces a pivot table):

```
select p.productclassid,
sum(case when s.country='USA' then 1 else 0 end) as 'USA',
sum(case when s.country='Canada' then 1 else 0 end) as
'Canada',
sum(case when s.country='Japan' then 1 else 0 end) as
'Japan',
sum(case when s.country='UK' then 1 else 0 end) as 'UK',
count(productid) TOTAAL
from product p join supplier s
on p.supplierid = s.supplierid
group by p.productclassid;
```

- Bundle xTreme
  - triggers extra oef. 1
  - triggers extra oef. 2
  - triggers extra oef. 3